Project 3: CNN for MNIST Classification

Tanmay Asthana
ECE Deptartment
North Carolina State University
Raleigh, NC, USA
tasthan@ncsu.edu

Nathan Starliper
ECE Deptartment
North Carolina State University
Raleigh, NC, USA
nstarli@ncsu.edu

Michael Patel
ECE Deptartment
North Carolina State University
Raleigh, NC, USA
mrpatel5@ncsu.edu

***Abstract*—Convolutional Neural Networks (CNNs) are an interesting genre unto themselves in the field of deep learning. CNNs are a particularly powerful tool for image classification. This paper examines CNNs by taking an in-depth consideration of different hyperparameters, activation functions, and regularization techniques. Based on the analytical results, we propose one such successful CNN model. All model design choices were made with regards to the MNIST dataset of handwritten digits.**

I. Introduction

The goal of deep learning is to discover hierarchical models that accurately represent data. The models seek to internalize the essence of data, which could be images, audio, etc. With the success of CNNs, deep learning models have flourished in the realm of classification.

Image classification involves the task of determining what class any given image belongs to. People, even young children, are able to perform this task quite easily, but computer models may not fair as much. For example, suppose an image classifier was given an image of an animal (a dog) and asked to determine whether it was a dog or a cat. Human beings could be quite certain that the image is a dog, but the classifier may only predict dog with 76% confidence.

One such application of image classification is to identify handwritten versions of the digits 0-9. This type of deep learning application has use cases for the postal service, automated homework grading, or text conversion to digital print.

II. Background

A Convolutional Neural Network (CNN) is a type of deep learning model that has proven successful in applications that require image classification. CNNs seek to find unique features in the data that aid in labeling an image with a specific class option. For example, a CNN may eventually learn that features such as eyes, lips and ears are necessary for determining whether an image is classified as human or not. CNNs have a successful track record with image classification.

An image consists of multiple pixels. If each pixel is taken as a single input to a rudimentary fully connected neural network, the size of the network will blow up exponentially and we will have to tune too many parameters. Eventually to classify things in more complicated colour images, such as buses, cars, trains etc., we run into problems with our accuracy and convergence rate. Too many weights and biases also makes the network highly complex and thus prone to overfitting. In fact, learning such difficult problems can become intractable for normal neural networks

III. CNN Design

A. Components of CNN

A CNN uses 2-D moving filters or templates instead of simple 1-D weight vectors. The filter is convolved with the 2-D pixel matrix of the input image sample. In a convolution operation, this moving filter would shuffle across each possible x and y co-ordinate combination of the input image to populate the output nodes. The simplicity in architecture arises because of below reasons:

- **Sparse Connections** - Unlike a fully connected network, not every input node (pixel) is connected to the output nodes.
- **Constant filter parameters / weights** Each filter has constant parameters. In other words, as the filter moves around the image the same weights are being applied. Each filter therefore performs a certain transformation across the whole image. A fully connected neural network generally has a different weight value for every connection

The output of the convolutional mapping is then passed through a non-linear activation function. This whole process is called **Feature Mapping** as a filter is trained to recognize specific features in the image. Multiple filters are used to detect multiple features and each such filter constitutes a channel.

After feature mapping, next layer is generally what is called a pooling layer. It basically combines multiple nodes of convolution layer in a specific way to get a single output node. By doing so,we try to achieve following goals:

- Further reduce the number of parameters in network by effectively performing a down-sampling
- Make feature detection more robust by making it more impervious to scale and orientation changes. Pooling acts as a generalizer of the lower level information and so enables us to move from high resolution data to lower resolution information

In our network we have performed max-pooling which basically spits out maximum out of all the nodes covered by the

max-pooling window at a time.
How a filter or maxpooling window traverses across the image is decided by the stride size which specifies how many steps the window should be moved in x- or y-direction at a time. To take care of border scenarios, if required, the image can be padded with 0's on it's sides. We can compute the size of the output layer in either x- or y-direction, as a function of the input size, W, the filter size F, the stride length, S, and the amount of zero padding, P applied on each side. It turns out to be $(W - F + 2P)/S + 1$. For example for a 7x7 input and a 3x3 filter with stride 1 and pad 0 we would get a 5x5 output. With stride 2 we would get a 3x3 output.
Finally a fully connected shallow neural network is connected after stacking together appropriate number of convolution and pooling layers. At the output of the convolutional-pooling layers we have moved from high resolution, low level data about the pixels to representations of objects within the image. The purpose of these final, fully connected layers is to make classifications regarding these objects.

B. Backpropagation through the network

Let's say in forward propagation, at certain convolution layer, we take a 3×3 matrix, X and a 2×2 matrix, W as input and get another matrix, a 2×2 matrix, h as output. As a filter traverses through A, each weight in W contributes to each pixel in h, as shown below:
$h_{11} = W_{11}X_{11} + W_{12}X_{12} + W_{21}X_{21} + W_{22}X_{22}$
$h_{12} = W_{11}X_{12} + W_{12}X_{13} + W_{21}X_{22} + W_{22}X_{23}$
$h_{11} = W_{11}X_{21} + W_{12}X_{22} + W_{21}X_{31} + W_{22}X_{32}$
$h_{11} = W_{11}X_{22} + W_{12}X_{23} + W_{21}X_{32} + W_{22}X_{33}$
Thus, any change in a weight in the filter will affect all the output pixels. Thus, all these changes add up to contribute to the final loss. Now, for implementing the back propagation step for the current layer, we can assume that we get ∂h as input (from the backward pass of the next layer) and our aim is to calculate ∂W and ∂X.Any change in a weight in the filter will affect all the output pixels. Thus, we can easily calculate the derivatives for W as follows:
$\partial W_{11} = \partial h_{11}X_{11} + \partial h_{12}X_{12} + \partial h_{21}X_{21} + \partial h_{22}X_{22}$
$\partial W_{12} = \partial h_{11}X_{12} + \partial h_{12}X_{13} + \partial h_{21}X_{22} + \partial h_{22}X_{23}$
$\partial W_{11} = \partial h_{11}X_{21} + \partial h_{12}X_{22} + \partial h_{21}X_{31} + \partial h_{22}X_{32}$
$\partial W_{11} = \partial h_{11}X_{22} + \partial h_{12}X_{23} + \partial h_{21}X_{32} + \partial h_{22}X_{33}$
Similarly, ∂X can be calculated.
In maxpool layer, there is no gradient with respect to non maximum values, since changing them slightly does not affect the output. Further the max is locally linear with slope 1, with respect to the input that actually achieves the max. Thus, the gradient from the next layer is passed back to only that node which achieved the max. All other nodes get zero gradient.

C. Optimization

We have used Adaptive Moment Estimation (Adam) method for updating our gradients in the mini-batch gradient descent. Adam keeps an exponentially decaying average of past squared gradients v_t and also an exponentially decaying average of past gradients m_t, similar to momentum.
We compute the decaying averages of past and past squared gradients m_t and v_t respectively as follows:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{1}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{2}$$

where g_t is the current estimate of the gradient. m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively. β_1 and $beta_2$ decide decay rate. As m_t and v_t are initialized as vectors of 0's, the authors of Adam observed that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. 1 and 2 are close to 1).

They counteract these biases by computing bias-corrected first and second moment estimates:

$$\hat{m_t} = \frac{m_t}{1 - \beta_1^t} \tag{3}$$

$$\hat{v_t} = \frac{v_t}{1 - \beta_2^t} \tag{4}$$

They then use these to update the network parameters (weights and biases), which yields the Adam update rule:

$\theta_{t+1} = \theta_t - \eta \frac{\hat{v_t}}{\sqrt{\hat{v_t} + \epsilon}}$
It can be shown that, Adam behaves like a heavy ball with friction, which thus prefers flat minima in the error surface.
We also use batch normalization and dropout for further optimization. Batch normalization provide any layer in a neural network with inputs that are zero mean/unit variance. Dropout is also an averaging technique. It reduces complex co-adaptations of multiple neurons on training data, by randomly deactivating some of them while training a mini-batch.

IV. Methodology

At a high level, the model takes in an image, and produces a predicted label (0-9) for that input image. The structure of our network involves a series of convolutional and pooling layers, followed by a dense layer, and lastly an output layer that predicts what digit the input image is. The dataset used for training, validation and testing is the MNIST dataset.

V. Implementation

A. Project Environment

Language: Python 3.6
Libraries: tensorflow, numpy, matplotlib, keras, pandas
Datasets: MNIST

B. Network Architecture

The structure of our neural network at a high level is depicted in Figure 1. The first convolutional layer takes in as input a batch of 28x28 images and uses a kernel size of 5x5 with 32 filters. The max pooling layer downsamples the output of the first convolutional layer, which is then followed by a second convolutional layer with 64 filters and kernel size of 5x5, and a second max pooling layer. Then, the tensors are flattened, and continue through a 256-neuron fully-connected layer before passing through a final output layer with 10 units representing the 10 class labels.

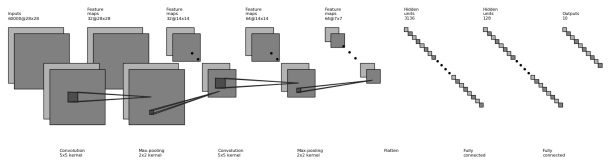

Fig. 1. Network Architecture

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 32)	832
batch_normalization (BatchNo	(None, 28, 28, 32)	128
max_pooling2d (MaxPooling2D)	(None, 14, 14, 32)	0
conv2d_1 (Conv2D)	(None, 14, 14, 64)	51264
batch_normalization_1 (Batch	(None, 14, 14, 64)	256
max_pooling2d_1 (MaxPooling2	(None, 7, 7, 64)	0
flatten (Flatten)	(None, 3136)	0
dense (Dense)	(None, 256)	803072
dropout (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 10)	2570

Total params: 858,122
Trainable params: 857,930
Non-trainable params: 192

Fig. 2. Summary of CNN Model

C. Cross-Validation

Cross-Validation was performed on the network to find the best hyper-parameters for the highest accuracy and lowest loss on a validation set. The validation set was extracted from the last 8000 images of the training set. The validation set was removed from the training set and was not used in training. The hyper-parameters were validated using the ReLU activation function. Table I shows the ranges of hyper-parameters validated with the best value found.

Figures 3, 4, 5, 6, 7, 8, 9, 10 show the results of validating the hyper-parameters. For each hyper-parameter, we plot the training and validation loss and accuracies against the parameter values. The images clearly show that accuracy is maximized and loss minimized for specific parameter values. We choose the parameter value that optimizes both of these values on the Validation set in order to minimize generalization error.

	Range Checked	Best Value
Learning Rate	0.001, 0.002, 0.005, 0.01	0.001
Batch Size	32, 64, 128, 256	64
Dropout Probability	0.1, 0.25, 0.5, 0.6, 0.75	0.6
Neurons in Dense Layer	64, 128, 200, 256	256

TABLE I

TABLE 1 SHOWS THE RANGES OF HYPER-PARAMETER VALUES VALIDATED AS WELL AS THE BEST VALUES FOUND

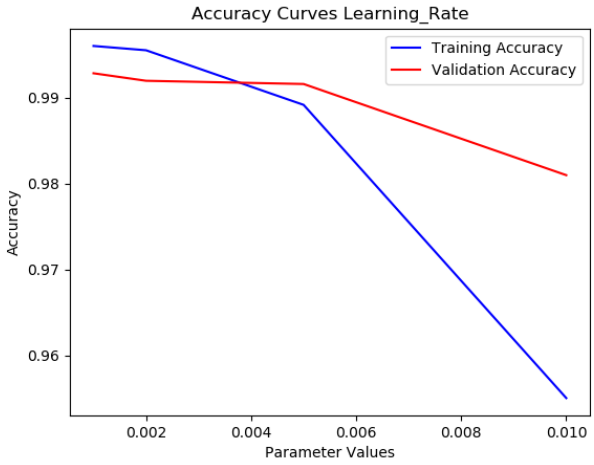

Fig. 3. Accuracy vs. Learning Rate

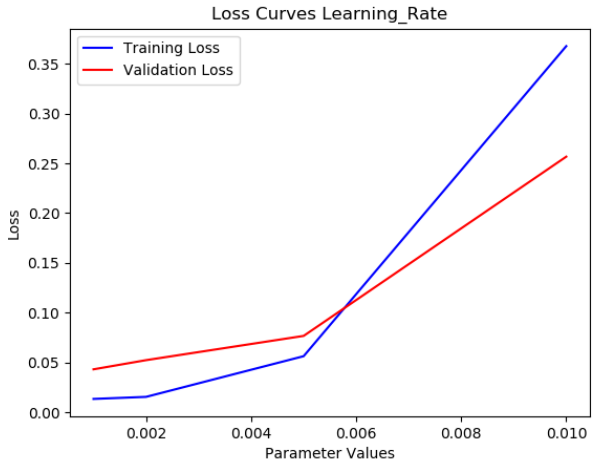

Fig. 4. Loss vs. Learning Rate

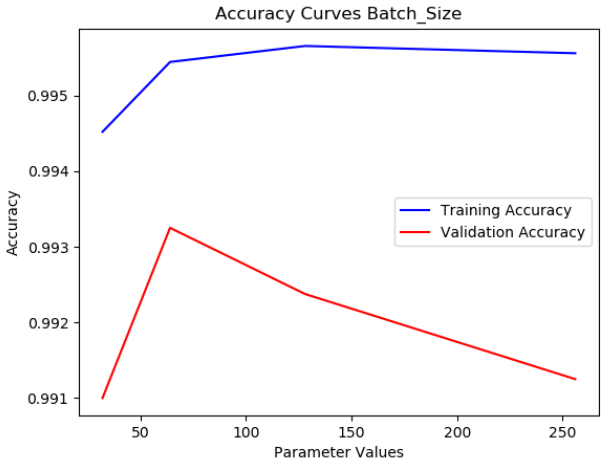

Fig. 5. Accuracy vs. Batch Size

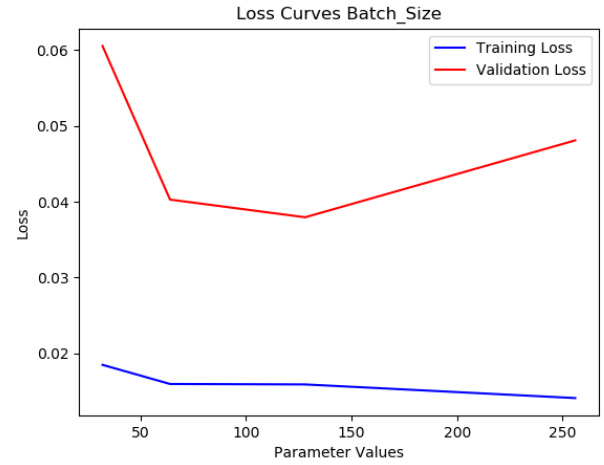

Fig. 6. Loss vs. Batch Size

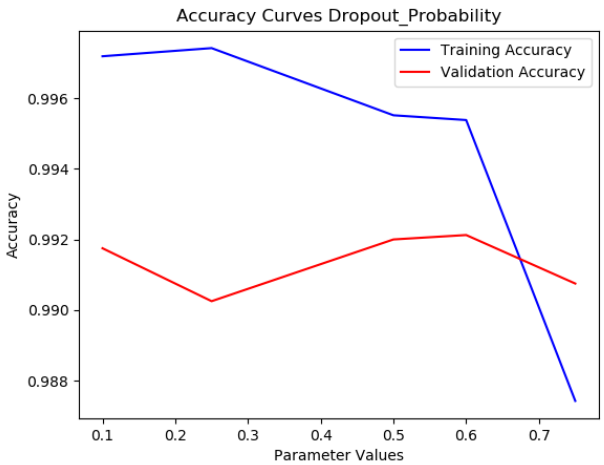

Fig. 7. Accuracy vs. Dropout Probability

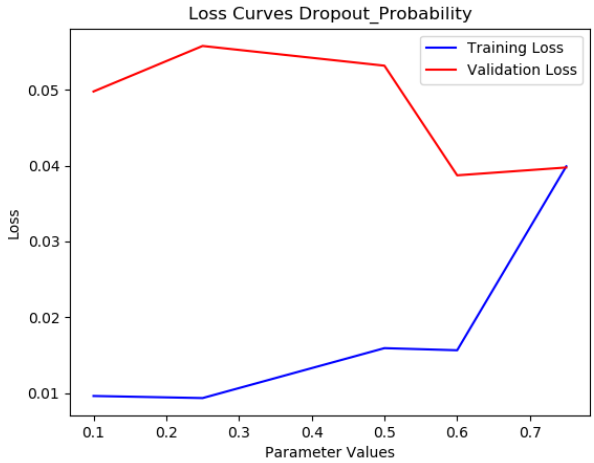

Fig. 8. Loss vs. Dropout Probability

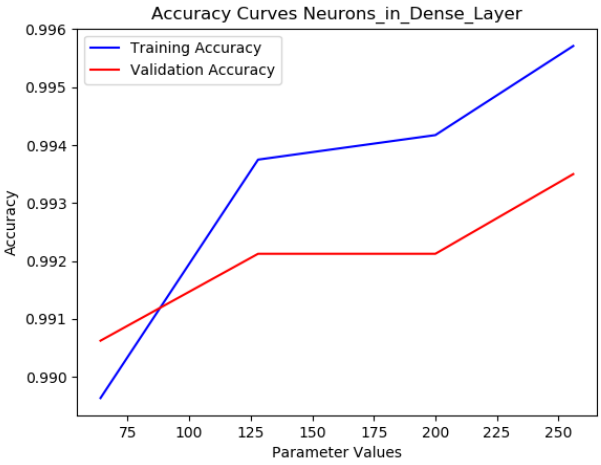

Fig. 9. Accuracy vs. Number of Neurons in Dense Layer

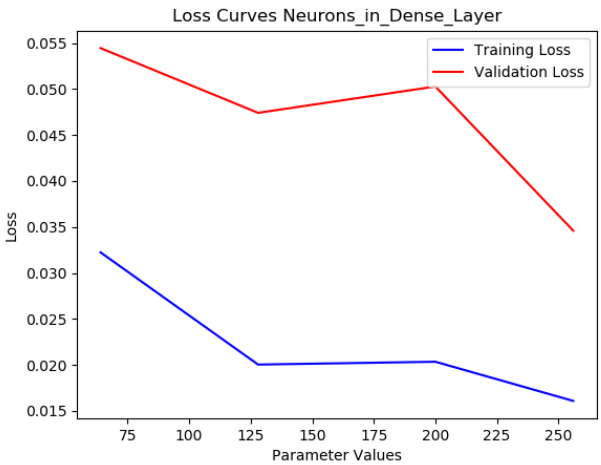

Fig. 10. Loss vs. Number of Neurons in Dense Layer

D. Batch Normalization

For our implementation, we added two layers that performed batch normalization. Each batch normalization layer was inserted after each convolutional layer. The effect of batch normalization seemed to be faster convergence during training. For example, with no batch normalization, we achieved 99.3% accuracy after approximately 20 epochs; with batch normalization, we achieved the same accuracy, but at 12 epochs.

E. Dropout

We experimented with the addition of a Dropout layer in between the 256-neuron Fully-connected layer and the final output layer. The range of experimented dropout probability values were [0.5, 0.6, 0.7, 0.8, 0.9]. The highest training, validation, and test accuracy was achieved with a dropout probability of 0.6.

VI. Results

Final testing loss and accuracy is shown in Table II. We tested the network using the following activation functions: ReLU, Tanh, and Sigmoid. The loss and accuracy curves on the training and validation sets with the different activation functions are seen in Figures 11, 12, 13, 14, 15, 16. Overall, we achieved the highest test accuracy at 99.39% using the ReLU activation function.

	Loss	Accuracy
ReLU	0.0324	0.9939
Tanh	0.0281	0.9912
Sigmoid	0.0286	0.9916

TABLE II
Loss and Accuracy Values for Different Activation Functions

It is also observed that Validation accuracy is higher than Training accuracy (and vice versa in case of loss). This ideally shouldn't be the case.

When accuracy is tested over a network trained over very few epochs the, the training loss is higher as it is being calculated over not fully mature network, while validation accuracy is tested over a fully mature network on which dropout regularization is already done. As number of epochs is increased training phase also experiences the averaging effect of dropout, which results in an increase in its accuracy and then we see that training accuracy is higher than validation accuracy as expected.

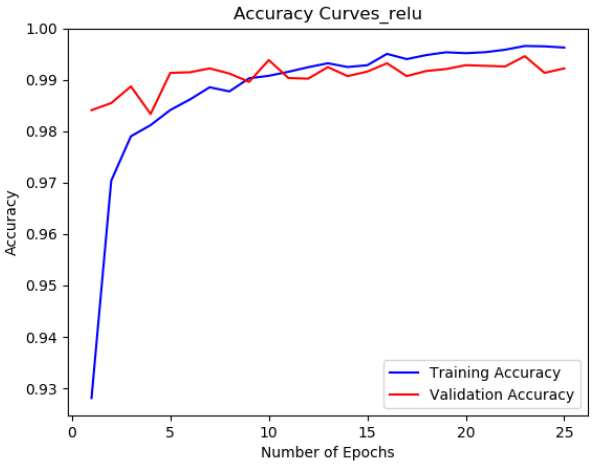

Fig. 11. ReLU Training/Validation Accuracy Curves

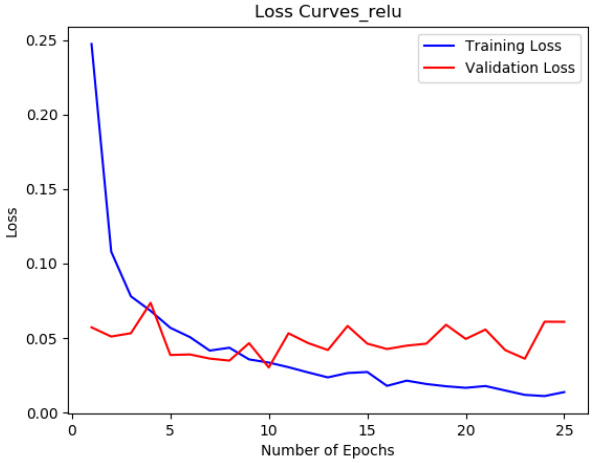

Fig. 12. ReLU Training/Validation Loss Curves

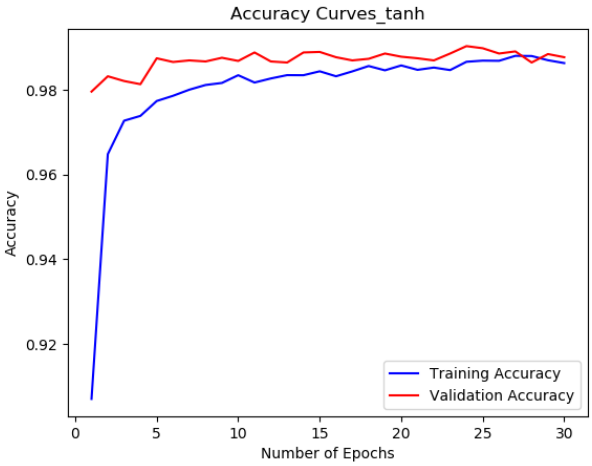

Fig. 13. Tanh Training/Validation Accuracy Curves

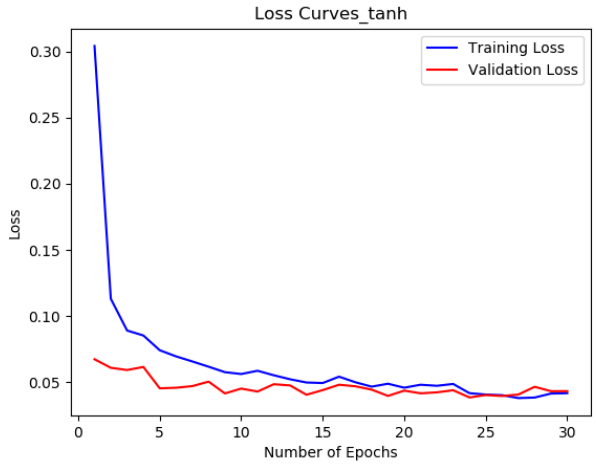

Fig. 14. Tanh Training/Validation Loss Curves

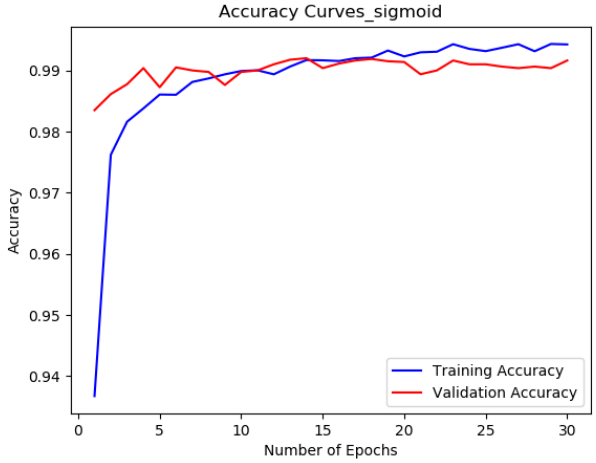

Fig. 15. Sigmoid Training/Validation Accuracy Curves

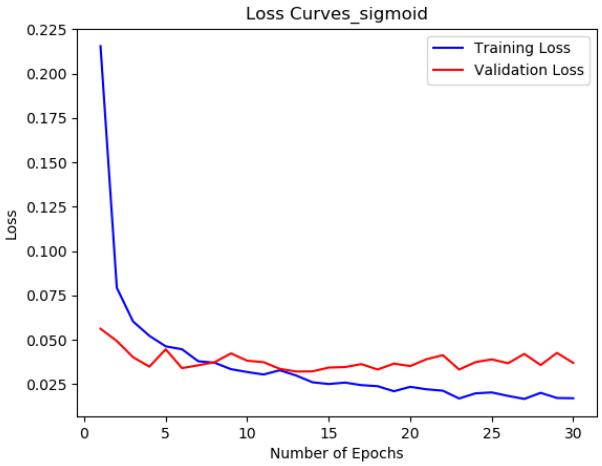

Fig. 16. Sigmoid Training/Validation Loss Curves

VII. Conclusion

Image classification using a CNN can be reasonably successful on the MNIST dataset with the implemented architecture and design decisions described above. By considering the trade-offs of different model choices with regards to accuracy and convergence time, our group learned quite a bit about the complexity and sensitivity involved with image classifiers. A future extension of this investigation would be to expand the scope of study to include parameter initialization.

References

[1] https://github.ncsu.edu/qge2/ece542-2018fall/tree/master/project/03
[2] http://ruder.io/optimizing-gradient-descent/